DOCUMENT RESUME

ED 388 302 IR 017 451

AUTHOR Weber, Gerhard; Mollenberg, Antje
TITLE ELM-PE: A Knowledge-based Programming Environment for Learning LISP.
PUB DATE 94
NOTE 7p.; In: Educational Multimedia and Hypermedia, 1994. Proceedings of ED-MEDIA 94--World Conference on Educational Multimedia and Hypermedia (Vancouver, British Columbia, Canada, June 25-30, 1994); see IR 017 359.
PUB TYPE Reports - Descriptive (141) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS Computer Literacy; *Computer Software Development; *Debugging (Computers); *Experiential Learning; Feedback; Foreign Countries; Instructional Effectiveness; *Learner Controlled Instruction; *Problem Solving; *Programming; Programming Languages; Skill Development

ABSTRACT

Novices in programming face many problems affecting their learning process and programming success. Learning to program includes using the programming environment, learning a programming language's syntax and semantics, understanding a problem and translating it into an executable plan, developing algorithms and programs, and testing and debugging programs. The knowledge-based programming environment ELM-PE is designed to support novices learning the programming language LISP. It has several features that are especially useful in problem solving in a new complex domain. These features are designed to avoid unnecessary mistakes, give immediate feedback, reduce memory load, support learner activity, and support example-based learning. Students can work on exercises with and without help from the system, to choose exercises on their own, to plan, program and debug function definitions, and to ask the system for help. The system remains passive as long as possible and offers advanced help only on demand. In an evaluation study, ELM-PE proved to be effective in facilitating learning. It was found that students who used the helping facilities often during the first lessons of programming performed well in a final exam; the more support available during the learning phase, the better students managed to solve the final programming problems when they were unable to use the advanced help system. (Contains 15 references.) (AEF)

ELM-PE
A Knowledge-based Programming Environment for Learning LISP

GERHARD WEBER AND ANTJE MÖLLENBERG
Department of Psychology
University of Trier, D-54286 Trier, Germany
E-Mail: weber@cogpsy.uni-trier.de

Abstract: The knowledge-based programming environment ELM-PE is designed to support novices learning the programming language LISP. It has several features that are especially useful in learning to solve problems in a new, complex domain. ELM-PE is an open, complete programming environment. It offers several facilities, e.g. visualization of executing programs, example-based programming, and cognitive diagnosis of program code, that go far beyond traditional ITSs. In an evaluation study, ELM-PE proved to facilitate learning. Students performing well in a final exam used the helping facilities more often during the first lessons of learning programming.

Novices in programming face many problems affecting their learning process and programming success. Learning to program includes acquiring knowledge and skills in various domains: Using the programming environment, learning a programming language's syntax and semantics, understanding a problem and translating it into an executable plan, developing algorithms and programs, and testing and debugging programs. In order to aid novices develop such abilities, various systems have been devised. These comprise Intelligent Tutoring Systems (ITS) (e.g., the CMU-LISP-Tutor (Anderson, Conrad, & Corbett, 1989), or BRIDGE (Bonar & Cunningham, 1988)) as well as knowledge-based programming environments and help systems (e.g., PROUST (Johnson, 1986), SCENT (McCalla & Greer, 1993), or ABSYNT (Möbus & Schröder, 1993)).

The knowledge-based programming environment ELM-PE is designed to support novices learning the programming language LISP. It has several features that are especially useful in learning to solve problems in a new, complex domain. These features are designed to avoid unnecessary mistakes, to give immediate feedback, to reduce working memory load, to support learner activity, and to support example-based learning. In contrast to an ITS like the CMU-LISP-Tutor, ELM-PE is an open, complete programming environment. Students are allowed to work on exercises with and without help from the system, to choose exercises on their own, to plan, to program, and to debug function definitions, and to ask the system for help if they need it. Students are allowed to produce programming errors, because students learn from debugging and from reflecting on their bugs and misconceptions. Therefore, debugging and helping facilities specifically designed for beginners are integrated into the programming environment. The system remains passive as long as possible and offers advanced help only on demand. Immediate feedback is given only for handling the graphical user interface, for syntactical errors, and if the student wants to work on a further exercise while the solution to the previous task is still incorrect.

The ELM-Programming Environment

ELM-PE can be run in three modes (LISTENER, EDITOR, and EXERCISE) among which the user can switch at any time. Modes differ according to the amount of support provided, so user control over the learning process is increased. For all modes passive as well as active help tools are implemented, using static as well as dynamic knowledge. Passive tools ensure the software-ergonomic principles of user definability and flexibility by leaving control to the student. Active tools interrupt students, but only when unnecessary errors (generally an attempt to use syntactically wrong input) or really serious errors (like switching to the next exercise when the actual solution is still incorrect) are detected that need to be dealt with immediately. This concerns the design

rules of tolerance and transparency. Basic to the system is the Listener mode. Each following mode inherits all features introduced in the previous mode. Users are thus able to choose the level of specific support they wish. The elaborated help system and the system's flexibility make us believe that ELM-PE exceeds most traditional programming environments in assisting beginners.

To demonstrate the scope of the system some of the facilities of ELM-PE supporting beginners learning a new programming language will be sketched briefly here:

Listener Mode

The system always starts in Listener mode which is the least supportive way of interacting with ELM-PE. In listener mode, syntax-structuring facilities are not activated. Transparency is enhanced by dimming all menus which are not applicable in the current status of the system and by system messages in the status line at the bottom of the screen. Messages comprise hints about what the user is expected to do next, short feedback if an action was rejected or information about the time the system will take to perform some task.

All messages from the system displayed in the status line and all error messages displayed in the LISP listener can be explained. This is very helpful for novices because in the beginning most error messages may be more confusing than useful. For common and very often occurring errors a short description is given why such errors can happen. When running in Editor mode, that part of the code where the run-time error occurred is also highlighted in the program window. For every object visible on the screen, further information may be retrieved via special mouse clicks, so passive on-line help is available during all stages of working with ELM-PE.

Editor Mode

In Editor mode, most supporting features of the system are available, e.g. programming in a syntax-driven structure editor, using examples, visualizing the evaluation of LISP-expressions, and displaying that part of code of a function definition where an error was detected.

Structure editor. Coding function definitions is supported by a syntax-driven LISP-editor (Köhne & Weber, 1987) to reduce syntax errors. In the program window of the structure editor, LISP-code can be produced by filling in slots for LISP-expressions with syntactically correct LISP-code, or with schemata for function calls containing slots for their arguments. These schemata can be accessed from a menu of LISP-functions, or from typing in the first part of a function call beginning with the function name. Additionally, code can be inserted from a buffer showing the most recently typed in or deleted expressions or by directly typing in complete LISP-expressions in the input window. The structure editor accepts as input only code that would result in syntactically correct LISP-expressions. Otherwise the input is rejected and an explanation of why the input would result in an incorrect expression is given.

Example-based programming. Examples of LISP-code can be displayed in a separate example window to support example-based learning (Neal, 1989). Examples can be selected from examples discussed in the learning materials of the LISP-course as well as from the set of function definitions the student has coded on its own. The example window has the same functionality as the program window of the structure editor, so LISP-expressions can be altered and parts of the code can be copied into the program window.

Visualization. One central principle in learning programming is to visualize the flow of data during evaluation of programs. This is supported by a stepper showing all evaluation steps. Corresponding expressions from self-defined function definitions – including definitions of sub-functions – are indicated in the program window. The stepper is supposed to help students envision dynamic program properties and to encourage active and self-directed learning. Additionally, those parts of the program code that are responsible for errors or are explained by the knowledge-based diagnosis can be highlighted in the program window.

Exercise Mode

The highest level of support is given in Exercise mode. When working in this mode, students select a programming problem to be solved from a menu of exercises for the lesson he or she is working on. The description of the task and some input-output-examples are displayed in a separate text window. In addition to all other facilities, special active as well as passive help can be given in this mode.

Dynamic analysis. If work at a task is aborted and the student has completed the code, i.e., no slots remain, the code is analyzed dynamically. A list of input-output descriptions is used to determine whether the student's

code returns the expected result. In case this is not true, the student is informed in a dialogue box and asked whether he or she still wants to terminate work. This is one of the few situations where the system is active by its own and will interrupt the student. Based on the input-output pairs the system used to test the function definition, the system is able to offer a function call with arguments leading to an incorrect result. Thus, students are encouraged to try to debug programs on their own. This is important because frequently students omit a thorough testing phase. In previous programming courses we observed that functions often were tested only once. If the arguments used do not reveal the error, students mistakenly think their solution was correct. Therefore, for this analysis to work the list of input-output descriptions must be complete so every error can be detected. The goal of this tool is to point out to students that extensive testing is an important skill as it can yield information about the type and location of errors.

Automatic cognitive diagnosis. Students can ask the system for help if they do not know how to complete coding the function definition, if they are not able to find an error in the code, or if they simply want to know whether their solution for the programming task will be correct. To offer students a complete explanation of errors and suboptimal solutions and to explain which plans have to be followed to solve the task correctly, a "cognitive diagnosis" was developed as a central "intelligent" component in the ELM programming environment. This diagnostic tool is based on the episodic learner model ELM (Weber, Bögelsack, & Wender 1993). The diagnosis results in an explanation how the program code submitted to the cognitive diagnosis could have been produced by the learner. This explanation is the basis of the system's hints to the learner what part of the code is incorrect, which plans must be followed to solve subgoals during problem solving and possibly partial solutions in the context of the learner's solution. Explanations are stored in the individual, episodic learner model. They can be used as the basis for retrieving structural and semantic analogies (Weber, 1991). As this component is central to the knowledge-based programming environment, it will be described in some more detail in the next section.

ELM: A Case-based Learning Model

ELM is a type of user or learner model that stores knowledge about the user (learner) in terms of a collection of episodes. In the sense of case-based learning, such episodes can be viewed as cases (Riesbeck & Schank, 1989). To construct the learner model, the code produced by a learner is analyzed in terms of the domain knowledge on the one side and a task description on the other side. This cognitive diagnosis results in a derivation tree of concepts and rules the learner might have used to solve the problem. These concepts and rules are instantiations of units from the knowledge base. The episodic learner model is made up of these instantiations and later generalizations based on them. To understand this form of episodic learner model, a short description of the knowledge representation and the diagnostic process will be given.

When using the environment, students code function definitions in a structured LISP-editor (Köhne & Weber, 1987), so their function code is at least syntactically correct. The cognitive analysis of the program code employs an explanation-based generalization (EBG) method (Mitchell, Keller, & Kedar-Cabelli, 1986). It starts with a task description related to higher concepts, plans, and schemata in the knowledge base. For every concept, a set of rules is stated describing different ways to solve the goal given by the plan of the concept. These rules are comparable to the notation of implementation methods for goals in PROUST (Johnson, 1986). There are "good", "suboptimal", and "buggy" rules explaining correct, correct but suboptimal, and incorrect solutions of the current goal or subgoal. The set of buggy rules is comparable to an error library in other ITSs (e.g., in the CMU-LISP-tutor (Anderson & Reiser, 1985). Applying a rule results in comparing the plan description to the corresponding part of the student's code. In the plan description, further concepts may be addressed. The cognitive diagnosis is called recursively until a function name, a parameter, or a constant is matched.

The cognitive diagnosis results in a derivation tree (Weber et al., 1993) built from all concepts and rules identified to explain the student's solution. This derivation tree is an explanation structure in the sense of EBG and is the basis for building up the episodic learner model. Concepts and rules addressed in the derivation tree are the basis for creating episodic frames. These frames are integrated into the knowledge base as instances of their concepts and rules. Slots in these instances refer to the context in which they were used (especially the current task), to the type of transformations of concepts, to the observed rules, and to the argument bindings.

In the next step, episodic frames are generalized. If an episodic frame is the first instance under a concept of the knowledge base, this single case is generalized from structural and semantic aspects in the data. This generalization mechanism is comparable to the single-case generalization in the EBG approach (Mitchell et al.,

1986). Additionally, similarity-based generalizations of data and plans can occur. Generalizations represent the class of types of plans and corresponding data which will be used in further cognitive analyses to interpret the student's code. With increasing knowledge about a particular learner, hierarchies of generalizations and instances are built under the concepts and rules of the knowledge base. They constitute the episodic learner model.

The ELM-approach differs from the case-based approach in SCENT (McCalla & Greer, 1933) in several aspects. In SCENT, pre-analyzed cases are stored as a whole with respect to a static granularity hierarchy that expresses aggregation and abstraction dimensions. These cases are used during analysis of the student's code to give detailed advice. In ELM only examples from the course materials are pre-analyzed and the resulting explanation structures are stored in the individual case-based learner model. Elements from the explanation structures are stored with respect to their corresponding concepts from the domain knowledge base, so cases are distributed in terms of instances of concepts. Generalization hierarchies of instances are built up from explanations of the program code that a student produced to solve programming problems. Therefore, generalization hierarchies reflect the process of knowledge acquisition for a particular student. These individual cases – or parts of them – can be used for two different purposes. On the one hand, episodic instances can be used during further analyses as shortcuts if the actual code and plan match corresponding patterns in episodic instances. On the other hand, cases can be used by an analogical component to show up similar examples and problems for reminding purposes (Weber, 1991).

Empirical Investigations

Programming Behavior

Students learning a programming language will be differently successful when finishing the introductory programming course, so we explored how students performing "well" vs. "poorly" on the programming tasks of the final exam after the recursion lessons differently used the help facilities of the programming environment and how they differed in their programming behavior. Good students in the practical test spent more time planning and debugging their solutions in the first lessons (Tab. 1). From the beginning they tested and evaluated their programs more frequently and used the help environment more often. Especially the cognitive diagnosis, but also the stepper and trace tools were called more frequently than by less successful students. This behavior is not restricted to the debugging phase of programming; good programmers actively seek additional information about their program using the help system while still planning their first solution. Looking at the two final lessons about recursion, one can see that this behavior changes to the opposite. Now, better students actually spend less time planning and debugging. They also do not use the help system as much as before. These effects also hold for the programming tasks on the final exam, where recursive programming exercises had to be solved. To explain these results, one must consider solution quality. Students who become good LISP programmers use the visualization of the evaluation of LISP expressions more often during most lessons and they seem to explore much more in the first phase of learning to program. These findings are in concordance with the interpretation of the self-explanation effect (Chi, Bassok, Lewis, Reimann, & Glaser, 1989). Often these students already have a correct solution but actively vary and test it a lot. This makes them gain more experience with different types of exercises on which they can rely later. In the lessons about recursion (lessons 4 - 6) where exercises tend to be more analogous they switch strategies and purposefully use their knowledge to program a correct solution quickly. At this stage of the course solutions are less frequently correct at the first try, but if they are correct, students are certain of understanding the problem and tackle the next exercise without further testing and exploring the solution as they did in the first lessons.

Comparing Programming Environments With Different Support

During development of ELM-PE we observed a substantial change concerning students' success in the final programming test. The more support is available during the learning phase, the better students manage to solve the final programming problems when they are cut off from the advanced help system. We want to illustrate this by comparing the results from students working in ELM-PE with the results from previous courses. In a first course (Others Group), no advanced help and no visualization was available and students were only supported by a structure LISP editor or a simple text editor matching parentheses. In a second course (TALUS Group), no cognitive diagnosis was available, but the complete program code could be analyzed by our version of the

Table 1

Tendencies of Differences in Means for Students Performing "well" (N = 10) vs. "poorly" (N = 8) in the Programming Tasks of the Final Exam.

	Lesson						
	1	2	3	4	5	6	Final tasks
Planning time	+	+	+	+	0	-	-
Debugging time	+	+	+	-	-	-	-
Calling help texts	+	+	+	+	-	-	-
Use of STEPPER	-	+	0	+	+	+	-
Use of TRACE	0	+	0	+	-	+	+
Calling cognitive diagnosis	0	+	+	+	0	+	not available

Note: + "good" students use/need more/longer
- "good" students use/need less/shorter
0 both groups same

TALUS debugger (Murray, 1988). In the final test two tasks were the same for all courses. One, "list-until-atom", requires a terminating case that was not practiced in the course material and as such is unknown to students. The other, "count-item", is of the car-cdr-recursion type and was very similar to problems in lesson 6. Using ELM-PE, most subjects worked on both problems and more than 75% succeeded in solving the tasks (Tab. 2). The task "count-item" was solved by 86.7 % and 80 % of all students from the ELM-PE and the TALUS Group, respectively. But, only 45 % of the students from the Others Group who tried to solve this problem were successful. The problem "list-until-atom" was hardest to solve in all courses. While 76.9% of all students from the ELM-PE Group who worked on this problem solved the task correctly, in the TALUS Group 61.5 % and in the Others Group only 16% were finally successful. Generally, the courses with the most support available to students on request have been the most successful in the final practical exam where support is temporarily removed from students.

Discussion

In several developmental steps, ELM-PE changed from a structure editor to an elaborated programming environment supporting beginners learning to program in LISP. As the system has not included an explicit tutorial component up to now, it is not a complete ITS when compared, for example, to Anderson's LISP-tutor (Anderson & Reiser, 1985; Anderson et al., 1989), but more a knowledge-based help system. Students are allowed to choose the number and sequence of exercises on their own, though it is recommended to work out all exercises in the given order. The system interrupts students only if serious errors (e.g., syntax errors and errors using the interface) happen. Students are allowed to produce programming errors, the intention being that students learn from debugging and from reflecting on their bugs and misconceptions. Therefore, debugging and helping facilities specifically designed for beginners are integrated into the programming environment. The system remains passive as long as possible and offers advanced help only on demand.

Results from empirical studies with ELM-PE support this point of view. Though there was less support from human tutors during exercises, we observed fewer drop-outs and better performance in our last version compared to previous versions. Students who perform better in a final programming test use the programming environment for planning and debugging in their first phase of learning LISP. Often, after having produced a first correct result to a programming problem, they explore alternative solutions and try to understand how the LISP interpreter evaluates function calls by using the stepper. Students profit from their freedom to explore the system, to debug programs, to ask the system for help, and to follow the LISP interpreter evaluating LISP expressions. This result fits in with the active learning view. Good programmers seem to learn LISP through exploring the environment early and connecting each relevant information to the programming situation at hand. Green & Gilhooly (1990) have named this "problem finding", the advantage being that when performance is measured without additional help (in the final test), they can rely on well-learned programming strategies. As Chi et al. (1989) observed, practice as a factor in learning success seems to be greatly influenced by individual

Table 2

Percentage of Correctly Solved Programming Tasks in the Final Test for Different Programming Environments.

	Task	
Programming Environment	Count-Item	List-Until-Atom
ELM-PE	86.7 % ($n = 30$)	76.9 % ($n = 26$)
TALUS-PE	80.0 % ($n = 15$)	61.5 % ($n = 13$)
Others	45.0 % ($n = 20$)	12.0 % ($n = 25$)

learning strategies. Self-monitoring of one's own programming is very important. Accordingly, a programming environment should enable students to organize their programming in their own fashion.

As a second result, the evaluation study indicates that AI-techniques as they are used in the TALUS-analysis in the previous version of the programming environment and especially in the cognitive diagnosis of ELM-PE enhance the success of a programming environment for beginners.

References

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science, 13*, 467-505.

Anderson, J. R., & Reiser, B. J. (1985). The LISP tutor. *Byte, 10*(4), 159-175.

Bonar, J., & Cunningham, R. (1988). BRIDGE: An intelligent tutor for thinking about programming. In J. Self (Ed.), *Artificial intelligence and human learning: Intelligent computer-aided instruction* (pp. 391-409). London: Chapman & Hall.

Chi, M. T. H., Bassok, M., Lewis, M., Reimann, P., & Glaser, R. (1989). Self-explanations: how students study and use examples in learning to solve problems. *Cognitive Science, 13*, 145-182.

Green, A. J. K., & Gilhooly, K. J. (1990). Individual differences and effective learning procedures: the case of statistical computing. *International Journal of Man-Machine Studies, 33*, 97-119.

Johnson, L. W. (1986). *Intention-based diagnosis of novice programming errors*. London: Pitman.

Köhne, A., & Weber, G. (1987). STRUEDI: a LISP-structure editor for novice programmers. In H. J. Bullinger, & B. Schackel (Eds.), *Human-Computer Interaction INTERACT '87* (pp. 125-129). Amsterdam: North-Holland.

McCalla, G. I., & Greer, J. E. (1993). Two and one-half approaches to helping novices learn recursion. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), *Cognitive models and intelligent environments for learning programming* (pp. 185-197). Berlin: Springer-Verlag.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: a unifying view. *Machine Learning, 1*, 47-80.

Möbus, C., & Schröder, O. (1993). The acquisition of functional planning and programming knowledge: Diagnosis, modeling, and user-adapted help. In G. Strube, & K. F. Wender (Eds.), *The cognitive psychology of knowledge. The German Wissenspsychologie project.* (pp. 233-262). Amsterdam: North-Holland.

Murray, W. R. (1988). *Automatic program debugging for intelligent tutoring systems*. London: Pitman.

Neal, L. R. (1989). A system for example-based learning. In K. Bice, & C. Lewis (Eds.), *Proceedings of Human Factors in Computing Systems, CHI'89,* (pp. 63-68). Reading, MA, Addison-Wesley.

Riesbeck, C. K., & Schank, R. C. (1989). *Inside case-based reasoning*. Hillsdale, NJ: Lawrence Erlbaum.

Weber, G. (1991). Explanation-based retrieval in a case-based learning model. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, (pp. 522-527), Chicago, IL.

Weber, G., Bögelsack, A., & Wender, K. F. (1993). When can individual student models be useful? In G. Strube, & K. F. Wender (Eds.), *The cognitive psychology of knowledge. The German Wissenspsychologie project.* (pp. 263-284). Amsterdam: North-Holland.

Acknowledgements

This work was supported by the German Science Foundation under Grant We 498/12.